Future Possibilities and Effectiveness of JIT from Elixir Code of Image Processing and Machine Learning into Native Code with SIMD Instructions

Susumu Yamazaki[1,a)]

Abstract: Nx is a multi-dimensional tensor library for Elixir with multi-staged compilation to the CPU or GPU, similar to NumPy and TensorFlow in Python. Nx is expected to be applied in image processing and machine learning. Code used by image processing and machine learning in C or C++ is often optimized for CPUs into native code with SIMD instructions. In this paper, we will show that native code with SIMD instructions is 1000x+ faster than equivalent Elixir code with Nx, to evaluate future possibilities and effectiveness of such code generation and optimization. Our future works are to implement and evaluate our proposal: a backend of Nx generating SIMD instructions by NIFs and/or BeamAsm using our compiler and/or OpenBLAS or cuBLAS.

Keywords: Code optimization, image processing, JIT, mechine learning, SIMD

1. Introduction

Nx [14] is a multi-dimensional tensor library for Elixir [13] with multi-staged compilation to the CPU or GPU, similar to NumPy [11] and TensorFlow [5] in Python [15]. Nx is expected to be applied in image processing and machine learning.

Code used by image processing and machine learning in C or C++ is often optimized for CPUs into native code with SIMD instructions or on GPU. Nx has such a backend, named EXLA [8], which calls Google XLA [6] and generates such code just in time or ahead of time.

We have developed Pelemay [1, 17–19, 23–26] [*1], a native compiler for Elixir, which generates SIMD instructions, and PelemayFp [20], a Fast parallel map function for Elixir. Especially, Pelemay can compile Elixir code into native code with SIMD instructions just in time, so we guess it can also be applied to Nx.

This paper will show that native code with SIMD instructions written by hands is more than 1000 times, 9 times, and 1.7 times faster than equivalent Elixir code with Nx, CPU native code generated by EXLA, and GPU code keep running on it generated by EXLA, respectively, to evaluate future possibilities and effectiveness of such code generation and optimization.

The rest of this paper consist of the following sections: Section 2 will describe prerequisite elemental technologies, Pelemay, Nx and BeamAsm. Next, Section 3 will propose our approach. Moreover, Section 4 will show the preliminary experiments of it. Last, Section 5 will summarize this paper and show

[1] Univ. of Kitakyushu, Kitakyushu, Fukuoka 808–0135, Japan
[a)] zacky@kitakyu-u.ac.jp

This paper is a compilation of an already published oral presentation of the same name [22].

[*1] Hastega is an old name of Pelemay and a name of magic spell that appeared in the Final Fantasy series.

```
defn softmax(t) do
  Nx.exp(t) / Nx.sum(Nx.exp(t))
end
```

Fig. 1: Sample code of Nx

the future works.

2. Nx

Nx [14] and EXLA [8] have not been officially released as of October 2021, and the pre-release version of their development has been released on GitHub.

Nx has the following nine functions, which are similar to NumPy [11] and TensorFlow [5]: Aggregates, backend, conversion, creation, element-wise, N-dim, shape, type, and others. Fig. 1 shows the sample code of Nx, which implements the Softmax function. In this code, `t` expresses a tensor. `Nx.exp(t)` returns a tensor, of which each element has a value of an exponential of the corresponding element of a tensor `t`. `Nx.sum(t)` returns the sum for a tensor `t`.`defn` defines a numerical function, a subset of Elixir tailored for numerical computation using Nx.

EXLA is a backend of Nx, which can compile numerical functions just in time or ahead of time for CPU, GPU and TPU. Another backend is `Nx.BinaryBackend`, which is an opaque backend written in pure Elixir that stores the data in Elixir's binaries. This is the default backend used by the Nx module. A programmer can define any backend of Nx.

3. Proposed Approach

Pelemay [1, 17–19, 23–26] has a function to compile Elixir code using `Enum.map` with simple arithmetic operations into native code with SIMD instructions using auto-vectorization of Clang and GCC, just in time. It also has the potential to optimize a series of operations into an integrated native code.

EXLA and Pelemay use Native Implemented Functions (NIFs) [3], functions that are implemented in C instead of Erlang [2] or Elixir. We found by experiments of a combination of Pelemay and PelemayFp [20] that a CPU-bound function implemented in NIFs is often slower than Elixir or Erlang code compiled into native code by BeamAsm [4], a JIT of Erlang. So then, we expect BeamAsm will be a good code generator for Pelemay and Nx and a good alternative FFI instead of NIFs [21].

Almost Nx functions can be compiled into operations of OpenBLAS [16] because they are linear algebra functions. Because OpenBLAS is the state-of-the-arts implementation to realize linear algebra functions in open-source software, it may be one of the fastest implementations of Nx. cuBLAS [10] is Basic Linear Algebra using CUDA [9], so it may also be one of the fastest implementations of Nx.

Thus, we propose implementing an Nx backend to compile pre-defined numerical functions into native code, including SIMD instructions. Its compilation technology may be SIMD code generation by auto-vectorization in GCC or Clang or our compiler, or BLAS code generation calling OpenBLAS or cuBLAS. Moreover, FFI technology may be NIFs or code generation by BeamAsm. Fig. 2 shows its structure.

Our approach is expected to achieve much efficiency of operations of tensors by Nx, by optimization of native code, and by elimination of overhead of FFI.

4. Preliminary Experiments

We conduct the preliminary experiments of our approach, the monochrome filter benchmarks[*2]. They process 65536 RGB 8-bit pixels into monochrome.

We implement the monochrome filter using Nx, C, hand-coded intrinsics of ARM NEON (with and without pipelining), and OpenCV [7], which uses OpenBLAS [16]. We implement that using C and hand-coded intrinsic using NIFs with transforming between `Nx.Tensor`, which expresses a matrix in Nx, and `uint8_t`, which is an 8-bit unsigned int. We also implement that using OpenCV using NIFs with transforming between `Nx.Tensor` and `cv::Mat`, which express a matrix in OpenCV.

The hand-coded intrinsics have a series of processes: Load multiple 3-element structures that have the 8bit RGB values to three registers, extend

[*2] This is available at `https://github.com/zacky1972/monochrome_filter`. The version used in the experiments is 0.2.0.

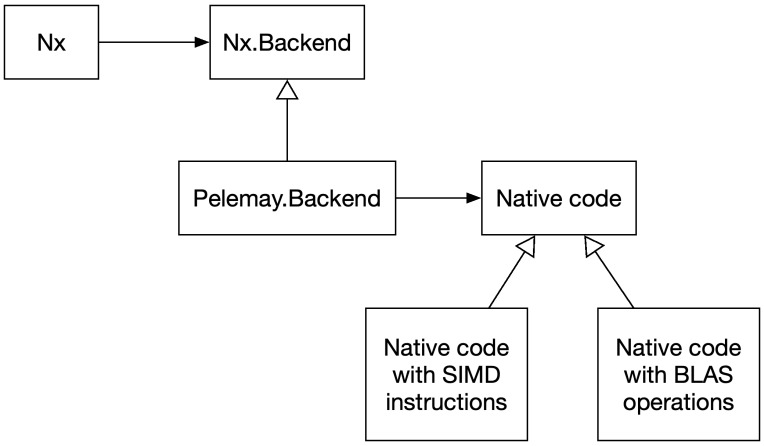

Fig. 2: The structure of the proposed approach

8bit into 16bit, extend 16bit into 32bit, convert an integer into a float, multiply, addition, convert a float into an integer, reduce 32bit into 16bit, reduce 32bit into 8bit, and store multiple 3-element structures from three registers.

The benchmarks are implemented using Benchee [12], which runs each kernel for a specified number of seconds after warming up, measures the iteration number, and shows the results of the iterations per second, average execution time, standard deviation, median of the execution time, 99th percentile, and memory usage.

We evaluate it on Apple M1 Mac and NVIDIA Jetson AGX Xavier shown in Table 1. We use them because their CPU is ARMv8, an architecture for which we implemented intrinsics of the benchmark.

The results of the average execution time of the benchmarks on Apple M1 Mac mini (M1, 2020) using Clang 13.0.0 and NVIDIA Jetson AGX Xavier using Clang 13.0.0 and GCC 11.2 are shown in Table 2, 3 and 4.

The source code of Nx is shown in Fig. 3. The items of xla are using EXLA, whose source code is the same as Nx. Next, Nx on Apple M1 Mac of 32bit is approximately the same as that of 16bit. Next, EXLA on CPU on Apple M1 Mac of 16bit and 32bit is 525 and 541 times faster than Nx, respectively. So the difference between 16bit and 32bit is tiny. The reason is probably that the operation in the case of 16bit includes converting 16bit into 32bit, operating in 32bit, and converting 32bit into 16bit.

Nx on NVIDIA Jetson AGX Xavier on 32bit is also approximately the same as that of 16bit. Next, EXLA on CPU on NVIDIA Jetson AGX Xavier on 16bit and 32bit is 352–400 and 381–410 times faster than Nx. Though the difference between 16bit and 32bit is tiny, the effectiveness of EXLA on Mac is 1.38 times larger than that on Jetson. The reason for the difference between Mac and Jetson is probably the difference in processor architecture.

EXLA on GPU on Jetson of 16bit and 32bit is 756–793 and 846–888 times faster than Nx, respectively. Next, EXLA on GPU with keeping the memory on the GPU on Jetson of 16bit and 32bit is 3297–3380 and 3321–3391 times faster than Nx, respectively. So the effectiveness of keeping is 4.08. Consequently, the key to speeding up using GPU is keeping. Moreover, GPU in the case of keeping is 8.68 times faster than CPU, so the effectiveness of GPU is this. These results and discussion explain the effectiveness of the existing approach.

The 16bit and 32bit NIFs on Mac are 2690 and 2689 times faster than Nx, respectively. Moreover, they on Jetson using Clang is 3652 and 3813 times faster than Nx. This effectiveness is approximately the same as GPU with keeping. Further, they on Jetson using GCC is 2676 and 5445 times faster than Nx. That of 32bit on Jetson using GCC is 1.43 times faster than that using Clang. The difference is caused by the effectiveness of the auto-vectorization of GCC. The difference is ununderstandable from the code because both Clang and GCC seem to generate SIMD instructions with auto-vectorization.

Table 1: Environment of the experiments

	Apple Mac mini (M1, 2020)	NVIDIA Jetson AGX Xavier
CPU	M1	NVIDIA Carmel Armv8.2
Cores of CPU	4 e-Cores and 4 p-Cores	8 Cores
CPU max clock	2.064GHz	2.2656GHz
fp16	available	N/A
GPU	M1 (8 cores)	512 NVIDIA CUDA Cores and 64 Tensor Cores
OS	macOS Big Sur 11.6	Linux kernel 2.9.253-tegra
Clang	13.0.0	13.0.0
GCC	N/A	11.2
Erlang	24.1.2	24.1.2
Elixir	1.12.3–otp–24	1.12.3–otp–24

Table 2: The results of the average execution time of the benchmarks using Clang 13.0.0 on Apple M1 Mac mini (M1, 2020)

	16bit (μs)	32bit (μs)
Nx	110237.64	112762.66
xla JIT CPU	209.82	208.39
NIF	40.98	41.94
NIF intrinsics	23.49	22.72
NIF intrinsics w/ pipeline	N/A	23.07
OpenCV CPU	N/A	12.90

Table 3: The results of the average execution time of the benchmarks using Clang 13.0.0 on NVIDIA Jetson AGX Xavier)

	16bit (μs)	32bit (μs)
Nx	422103.85	442101.99
xla JIT CPU	1197.53	1161.17
xla JIT GPU	558.63	522.83
xla JIT GPU keep	128.03	133.11
NIF	115.59	115.94
NIF intrinsics	N/A	79.81
NIF intrinsics w/ pipeline	N/A	79.65
OpenCV CPU	N/A	59.64
OpenCV GPU	N/A	72.74

Table 4: The results of the average execution time of the benchmarks using GCC 11.2 on NVIDIA Jetson AGX Xavier)

	16bit (μs)	32bit (μs)
Nx	421708.27	435458.64
xla JIT CPU	1053.86	1060.92
xla JIT GPU	531.67	490.52
xla JIT GPU keep	124.77	128.41
NIF	157.60	79.97
NIF intrinsics	N/A	80.28
NIF intrinsics w/ pipeline	N/A	82.99
OpenCV CPU	N/A	59.31
OpenCV GPU	N/A	70.96

Table 5 shows the part of the result of LLVM Machine Code Analyzer (`llvm-mca`) of them on Apple Mac mini (M1, 2020) on Clang 13.0.0 and GCC 11.2 compiled by NVIDIA Jetson AGX Xavier[*3].

[*3] The reason that we use Mac instead of Jetson is that we cannot prepare LLVM Machine Code Analyzer on Jetson.

Table 5: The part of the results of LLVM Machine Code Analyzer of NIF on Apple Mac mini (M1, 2020) on Clang 13.0.0 and GCC 11.2 compiled by NVIDIA Jetson AGX Xavier

	Clang 13.0.0	GCC 11.2
Iterations	100	100
Instruction	43,400	42,600
Total Cycles	17,139	23,545
Total uOps	51,600	456,000
Dispatch Width	6	6
IPC	2.53	1.94
Block RThroughput	93.0	76.0

Unlike the execution time, the IPC of Clang is 1.3 times larger than that of GCC. Moreover, IPC corresponds to the utilization of ALU. Then, we guess that IPC from LLVM Machine Code Analyzer may be an index of optimization by Clang. Because the results of IPC are opposed to the actual execution time in the case of our benchmarks, Clang cannot compile them into efficient native code.

The code of NIF intrinsics of 16bit cannot run on Jetson because NVIDIA Carmel Armv8.2 does not support fp16 NEON instructions (See Table 1). NIF intrinsics of 16bit and 32bit on Mac are 4693 and 4963 times faster than Nx. Moreover, these are 1.74 and 1.85 times faster than NIF. These are the difference between auto-vectorization by Clang and intrinsics. Actually, NIF with auto-vectorization by GCC is approximately the same as NIF intrinsics. NIF intrinsics of 16bit and 32bit on Mac are 8.93 and 9.17 times faster than xla JIT CPU. NIF intrinsics of 32bit compiled by Clang and GCC on Jetson are 5539 and 5445 times faster than Nx, respectively. Though that compiled by Clang is 1.45 times faster than NIF, that compiled by GCC is as fast as NIF compiled by GCC. NIF intrinsics of

32bit on Jetson are 13.9 times faster than xla JIT CPU. These show the potential in case that we will implement simple code generation, including SIMD instructions. It is also 1.7 times faster than xla JIT GPU keep.

Next, the execution time of NIF intrinsics with the pipeline is approximately the same as that of NIF intrinsics. These show that simple software pipelining may not be effective to M1 and Carmel Armv8.2, CPUs with out-of-order execution.

OpenCV CPU on Mac and Jetson is 8741 and 6108 times faster than Nx, respectively. Moreover, they are 1.76 and 1.35 times faster than NIF intrinsics, respectively. Therefore, they may be the best optimization case of ARM CPU. OpenCV uses OpenBLAS to calculate matrix operations. Practically, an operation on Nx can be compiled into operations on OpenBLAS. THerefore, to estimate the best optimization case of Nx, we should evaluate such operations on OpenBLAS. This evaluation remains as future work.

OpenCV GPU on Jetson is 1.21 times slower than OpenCV CPU. The reason for this is that the image size of this benchmark is relatively small, which is only 65536 pixels.

5. Summary and Future Works

We proposed implementing an Nx backend to compile pre-defined numerical functions into native code, including SIMD instructions. Its compilation technology may be SIMD code generation by auto-vectorization in GCC or Clang or our compiler, or BLAS code generation calling OpenBLAS or cuBLAS. Moreover, FFI technology may be NIFs or code generation by BeamAsm. Our approach is expected to achieve much efficiency of operations of tensors by Nx, by optimization of native code, and by elimination of overhead of FFI. We also showed that our approach might hopefully be competitive against EXLA by conducting the preliminary experiments.

Our future works are to implement and evaluate our proposal: a backend of Nx generating SIMD instructions by NIFs and/or BeamAsm using our compiler and/or OpenBLAS or cuBLAS.

Acknowledgments This research is supported by Adaptable and Seamless Technology transfer Program through Target-driven R&D (A-STEP) from Japan Science and Technology Agency (JST) Grant Number JPMJTM20H1.

References

[1] Abe, R., Yamazaki, S. and Takase, H.: Accelerate Elixir Application by Generating OpenCL Code, *IPSJ Special Interest Group on Programming 130th Meeting*, Information Processing Society of Japan (IPSJ) (2020). in Japanese.

[2] Ericsson: Erlang Programming Language (1998). `https://www.erlang.org`.

[3] Ericsson AB.: NIFs: Erlang Interoperability Tutorial (2000). `http://erlang.org/doc/tutorial/nif.html`.

[4] Ericsson AB.: BeamAsm, the Erlang JIT (2020). `http://erlang.org/documentation/doc-12.0-rc1/erts-12.0/doc/html/BeamAsm.html`.

[5] Google: TensorFlow: An Open Source Machine Learning Framework for Everyone (2017). `https://www.tensorflow.org`.

[6] Google: PyTorch/XLA: Enabling PyTorch on Google TPU (2018). `https://github.com/pytorch/xla`.

[7] Intel Research: OpenCV: Open Source Computer Vision Library (1999). `https://opencv.org`.

[8] Moriarity, S.: Elixir client for Google's XLA (Accelerated Linear Algebra). (2020). `https://github.com/elixir-nx/nx`.

[9] NVIDIA: CUDA: Develop, Optimize and Deploy GPU-accelerated Apps (2006). `https://developer.nvidia.com/cuda-toolkit`.

[10] NVIDIA: cuBLAS: Basic Linear Algebra on NVIDIA GPUs (2017). `https://developer.nvidia.com/cublas`.

[11] Oliphant, T.: NumPy: NumPy is the fundamental package for scientific computing with Python. (2005). `http://www.numpy.org`.

[12] Pfeiffer, T.: Benchee: Easy and extensible benchmarking in Elixir providing you with lots of statistics! (2016). `https://github.com/bencheeorg/benchee`.

[13] Valim, J.: Elixir: Elixir is a dynamic, functional language designed for building scalable and maintainable applications. (2013). `https://elixir-lang.org`.

[14] Valim, J.: Nx: Multi-dimensional arrays (tensors) and numerical definitions for Elixir (2020). `https://github.com/elixir-nx/nx`.

[15] van Rossum, G.: Python: Python is a programming language that lets you work quickly and integrate systems more effectively. (1991). https://www.python.org.

[16] Xianyi, Z. and Kroeker, M.: OpenBLAS: OpenBLAS is an optimized Basic Linear Algebra Subprograms (BLAS) library based on GotoBLAS2 1.13 BSD version. (2011). `http://www.openblas.net`.

[17] Yamazaki, S.: Hastega: Challenge for GPGPU on Elixir, *Lonestar ElixirConf 2019, Austin, TX, USA* (2019). The movie of this presentation is available at `https://youtu.be/lypqlGlK1So`.

[18] Yamazaki, S.: Application of Pelemay Parallel Programming System to ARM Architecture and Nerves IoT platform, Vol. 2020-EMB-54, pp. 1–8 (2020). `http://id.nii.ac.jp/1001/00204856/`, in Japanese.

[19] Yamazaki, S.: Pelemay Updates, *ElixirConf*

EU Virtual 2020 (2020). presentation is available at `https://speakerdeck.com/zacky1972/pelemay-updates`.

[20] Yamazaki, S.: PelemayFp: An efficient parallelization library for Elixir based on skeletons for data parallelism, *IPSJ Special Interest Group on Programming 133th Meeting* (PRO, I., ed.) (2020).

[21] Yamazaki, S.: Expected Application of BeamAsm, *Erlang 2021, Virtual* (Aronis, S., ed.) (2021). This is a lightening talk. The presentation slides are available at `https://speakerdeck.com/zacky1972/expected-application-of-beamasm`.

[22] Yamazaki, S.: Future Possibilities and Effectiveness of JIT from Elixir Code of Image Processing and Machine Learning into Native Code with SIMD Instructions, *IPSJ Special Interest Group on Programming 136th Meeting*, Information Processing Society of Japan (IPSJ) (2021).

[23] Yamazaki, S. and Hisae, Y.: Return of Wabi-Sabi: Hastega Will Bring More and More Computational Power to Elixir, *ElixirConf US 2019, Denver, CO, USA* (2019). The movie and of the slides of this presentation are available at `https://youtu.be/uCkPyfFhPxI` and `https://speakerdeck.com/zacky1972/return-of-wabi-sabi-hastega-will-bring-more-and-more-computational-power-to-elixir`, respectively.

[24] Yamazaki, S. and Hisae, Y.: SumMag: Design and Implementation of an Analyzer an Extension Mechanism by Meta-programming Using Elixir Macros, *Information Processing Society of Japan. Transactions on programming*, Vol. 12, No. 3, pp. 7–7 (2019). `https://ci.nii.ac.jp/naid/170000180471/`, in Japanese.

[25] Yamazaki, S. and Hisae, Y.: Performance Evaluation of SIMD Parallelization for Elixir Based on Skeletons for Data Parallelism, *IPSJ Special Interest Group on Programming 130th Meeting*, Information Processing Society of Japan (IPSJ) (2020). in Japanese.

[26] Yamazaki, S., Mori, M., Ueno, Y. and Takase, H.: A Method Using GPGPU for Super-Parallelization in Elixir Programming (in Japanese), *Proceedings of Summer United Workshops on Parallel, Distributed and Cooperative Processing 2018 (SWoPP 2018)*, IPSJ Special Interest Group on Programming, Tokyo, Japan, Information Processing Society of Japan (IPSJ) (2018). in Japanese.

```
defmodule MonochromeFilter do
  import Nx.Defn

  defn transpose_vector(vector) do
    Nx.reshape(
      vector,
      {Nx.size(vector), 1}
    )
  end

  defn broadcast_vector(
    vector,
    shape_tensor
  ) do
    vector
    |> transpose_vector()
    |> Nx.broadcast(shape_tensor)
  end

  defn monochrome_filter_32(pixel) do
    assert_shape_pattern pixel, {_, 3}

    mono = Nx.tensor(
      [0.299, 0.587, 0.114],
      type: {:f, 32}
    )

    pixel_m = Nx.dot(pixel, mono)

    broadcast_vector(pixel_m, pixel)
    |> Nx.round()
    |> Nx.as_type({:u, 8})
  end

  defn monochrome_filter_16(pixel) do
    assert_shape_pattern pixel, {_, 3}

    mono = Nx.tensor(
      [0.299, 0.587, 0.114],
      type: {:f, 16}
    )

    pixel_m = Nx.dot(pixel, mono)

    broadcast_vector(pixel_m, pixel)
    |> Nx.round()
    |> Nx.as_type({:u, 8})
  end

  defn init_pixel() do
    Nx.tensor(
      [0x9f, 0x5a, 0xae],
      type: {:u, 8}
    )
    |> broadcast_vector(
      Nx.iota({3, 65536})
    )
    |> Nx.transpose()
  end

  defn init_random_pixel() do
    Nx.random_uniform({65536, 3})
    |> Nx.multiply(255)
    |> Nx.round()
    |> Nx.as_type({:u, 8})
  end
end
```

Fig. 3: Source code of the monochrome filter in Elixir and Nx